Locking Aspects in Multithreaded MPI Implementations

ABDELHALIM AMER, Argonne National Laboratory
HUIWEI LU, Argonne National Laboratory
YANJIE WEI, Shenzhen Institute of Advanced Technologies, Chinese Academy of Sciences
JEFF HAMMOND, Intel
SATOSHI MATSUOKA, Tokyo Institute of Technology
PAVAN BALAJI, Argonne National Laboratory

MPI implementations rely mostly on locking to provide thread safety and comply with the MPI standard requirements. Yet despite the large body of literature that targets improving lock scalability and fine-grained synchronization, little is known about the arbitration aspect of locking and its effect on MPI implementations. In this paper, we provide an in-depth investigation of the correlation between locking and progress in an MPI runtime. We found that communication progress can be severely hindered when unbounded lock monopolization takes place without yielding work. This situation is common in practice when threads polling for communication progress monopolize an unfair lock, such as a Pthread mutex. We show that FIFO locks can mitigate this issue by circulating the lock among contending threads. In addition, we demonstrate that an adaptive lock that reduces the inference stemming from waiting threads can yield better progress by promoting threads with a higher work potential. Through an extensive comparative evolution of the different locks using benchmarks and applications, we show substantial improvements over the commonly used Pthread mutex approach.

CCS Concepts: •**Computing methodologies** → **Massively parallel algorithms; Concurrent algorithms;** •**Software and its engineering** → **Parallel programming languages; Distributed programming languages; Concurrent programming languages;**

Additional Key Words and Phrases: MPI, threads, runtime contention, critical section

ACM Reference Format:
Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Jeff Hammond, Satoshi Matsuoka, Pavan Balaji, 2016. Locking Aspects in Multithreaded MPI Implementations *ACM Trans. Parallel Comput.* X, X, Article 1 (June 2016), 27 pages.
DOI: 0000001.0000001

1. INTRODUCTION

MPI has been the de facto standard for programming high-performance computing (HPC) systems for more than two decades. With the advent of multi- and many-core systems, however, relying purely on MPI to handle both inter- and intranode parallelism is being questioned. Indeed, per-core resources (e.g., memory capacity and network endpoints) are becoming scarcer, implying that programming systems that promote collaboration within the same node are more desirable than those inherently competitive. Unfortunately, MPI's

This work was supported by JSPS KAKENHI Grant Number 23220003 and by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.
Authors addresses: Abdelhalim Amer, Argonne National Laboratory, USA (aamer@anl.gov); Huiwei Lu, (Current address) Tencent China (huiweilv@tencent.com); Yanji Wei, Shenzhen Institute of Advanced Technologies, Chinese Academy of Sciences, China (yj.wei@siat.ac.cn); Jeff Hammond, Intel USA (jeff_hammond@acm.org); Satoshi Matsuoka, Tokyo Institute of Technology, Japan (matsu@is.titech.ac.jp); Pavan Balaji, Argonne National Laboratory, USA (balaji@anl.gov).

DOI: 0000001.0000001

default private address-space model, where processes view resources as dedicated, falls into the latter category. As a result, applications, libraries, and languages centred around MPI are moving toward hybrid models that exploit MPI for coarse-grained parallelism and another programming system more suitable for shared memory. Among these models, MPI+threads is predominant, with MPI+OpenMP being the most widely used hybrid programming model.

Sharing the same address space, however, comes with costs associated with synchronization to maintain data consistency. Thus, avoiding or reducing synchronization costs is key to exploiting the collaborative behavior of threads while reducing or eliminating destructive behaviors. To reduce unnecessary overhead, the MPI standard offers applications different degrees of interoperability between threads and MPI. If an application requires concurrent multithreaded access, however, thread-compliant MPI implementations have to guarantee thread safety. This is ensured by combining critical sections and lock-free synchronization in virtually all production MPI implementations.

A major bottleneck for concurrent algorithms is the contention that arises in the presence of synchronized accesses to shared data. Such a bottleneck is often mitigated through two orthogonal approaches. On the one hand, by reducing the granularity of the synchronized region, whether through fine-grained locking or lock-free synchronization, contention can be lowered substantially. Most MPI implementations satisfy thread compliance by using coarse-grained critical sections because of their simplicity, although we explored the use of fine-grained locking as well [Balaji et al. 2010; Dózsa et al. 2010]. On the other hand, by improving the scalability of the synchronization mechanism—by reducing cache-coherency traffic and the number of atomic operations and memory barriers, for instance—the serialization and the likelihood of contention can also be decreased. In particular, recent advances in locking gave birth to more scalable locks on modern multi-core hardware, such as hierarchical and cohort locks [Chabbi et al. 2015].

Despite the large body of literature on granularity and scalability optimizations, an orthogonal dimension of the problem has often been ignored: *arbitration* of the synchronized accesses. Although existing works have brought insight into arbitration issues when sharing resources, such as starvation and priority inversion problems, little research has been conducted on the role of arbitration in improving the progress of a concurrent system. It is difficult to know a priori which thread will contribute most to the progress of a system because several factors are involved, including the length of the critical section, the lock hand-off latency, and fairness and load-balancing implications. Therefor, an optimal and generic synchronization method is not practical. We argue, however, that for a given system it is feasible to bias the arbitration to favor execution paths that contribute better to the overall progress.

In this paper we explore the correlation between arbitration and progress in the context of a multithreaded MPI runtime as the target concurrent system. We first analyze the effect on communication progress of using the system-provided Pthread mutex to implement critical sections, an approach adopted by several MPI implementations on HPC systems. Our analysis shows that severe unbounded lock monopolization can take place within the runtime and hinder the overall progress. This stems from a combination of the Pthread mutex, characterized by a competitive hand-off model biased by the memory hierarchy and kernel scheduling, and waiting threads polling for communication progress interfering with other threads, potentially having higher chances of advancing the system.

We then analyze the effect of introducing fairness by using first-in, first-out (FIFO) locks. We show that this approach can reduce the bias and improve communication progress by circulating the lock among contending threads. In addition, we demonstrate that in order to retain the arbitration benefits and achieve good communication progress, a combination with a low hand-off latency is essential, where a queuing lock showed the highest performance. Furthermore, we investigate an approach that biases the arbitration towards

Table I. Platform Specifications

Machine Specification	Thing from JLSE	Fusion from LCRC
Intel Architecture	Haswell	Nehalem
Processor	Xeon E5-2699 v3	Xeon E5540
Clock frequency	2.3 GHz	2.6 GHz
Number of sockets	2	2
Number of NUMA nodes	4*	2
Cores per NUMA node	9	4
L3 Size	23 MB	8192 KB
L2 Size	256 KB	256 KB
Number of nodes	8	310
Interconnect	Mellanox FDR	Mellanox QDR
Compilers	Intel 15.0.3	GNU 4.7.2
Linux Kernel	3.10.0-327.el7	2.6.18-402.el5
Glibc	2.17	2.5

* Two NUMA nodes per socket in this processor is a result of cluster-on-die (COD) mode, which is a BIOS option. The more common option is a single NUMA domain per socket.

reducing interference from waiting threads. This approach obstructs threads waiting for progress and promotes execution paths that have higher work potential using a two-level priority lock.

We extensively evaluate the various locking strategies on multicore InfiniBand clusters with microbenchmarks, stencil and graph kernels, a particle transport proxy application, and a genome assembly application. Results show that improvements of 50–100% compared with Pthread mutex are common, up to 10-fold improvements were observed with the particle transport proxy application, and that obstructing waiting threads can be beneficial or add only negligible overhead when the interference from waiting threads is not significant when using FIFO locks.

This paper is organized as follows. In Section 2 we briefly describe the different testing platforms used throughout the paper. On Section 3 we introduce the thread-safety challenges in MPI in general; present related work on improving the threading support in MPI implementations; and describe thread-safety measures in MPICH [Amer et al. 2015a], the most widely used MPI implementation. In Section 4 we analyze the arbitration and hand-off latency of the most popular locks. In Section 5 we discuss the lock monopolization effect on communication progress. In Section 6, we describe and analyze the effect of using FIFO locks and our custom priority lock. In Section 7 we compare the different locking methods using microbenchmarks, 3D stencil and graph traversal kernels, a particle transport proxy application, and a genome assembly application. Additional discussion of the impact of the proposed work is presented in Section 8. In Section 9 we present concluding remarks and ideas for future work.

2. TESTING PLATFORMS

Our analysis and evaluation were conducted on two commodity multicore InfiniBand clusters, as detailed in Table I. The nodes on both machines are based on Intel dual-socket processors interconnected with a Mellanox InfiniBand fabric. The first, Thing, is a small-scale machine located at the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory (ANL). It uses recent hardware, namely, 36 Haswell cores with an FDR interconnect. We use it to perform single-node and small scale MPI communication analysis and evaluation. The second, Fusion, is a larger-scale machine located at the Laboratory Computing Resource Center (LCRC) at ANL as well. It has older-generation hardware and is used primarily for large-scale runs. In the remainder of this paper, we refer to these machines by the processor architecture names, that is, Haswell and Nehalem. Our baseline MPI implementation is the recent MPICH 3.2 built on top of the Mellanox Messaging Accelerator (MXM) interface version 3.4.

3. THREAD SAFETY AND MPI

Before we discuss the locking aspects in MPI implementations, we provide background about thread safety in the MPI standard. We then present related work on improving the threading support in MPI, and we discuss the design and implementation of thread safety in MPICH.

3.1. MPI Requirements for Thread Safety

Depending on the degree of interoperation between application threads and MPI, an MPI implementation may not be required to provide a fully fledged thread-safety support. For instance, if at most one thread is guaranteed to perform MPI calls, the MPI implementation may relax thread synchronization within the runtime in order to reduce unnecessary overheads. For this purpose, the MPI standard defines four levels of threading support: `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. These levels are listed with an increasing degree of threading support corresponding to an increasing degree of programming flexibility as well as potentially higher thread-safety overheads. Using these levels, an application informs the MPI runtime at initialization of the desired threading support. In particular, this paper focuses on the most flexible level, `MPI_THREAD_MULTIPLE`, which allows concurrent multithreaded MPI calls.

The MPI 3.1 standard does not allow a thread to have its own rank; instead, the rank is assigned at the MPI process level, which can internally be multithreaded. Thus, threads are not separately addressable, and communication occurs at the process level. Since any thread can consume a message sent to a target MPI process, threads share all communication resources, such as message and request queues. Furthermore, the MPI's progress semantics require that a thread blocked inside the MPI runtime not block the progress of other threads. Another restriction applies to message ordering, where messages have to be matched in the order they were issued, regardless of threading. These restrictions imply that substantial sharing is required for a multitheaded MPI implementation. As a result, most MPI implementations rely on coarse-grained critical sections for simplicity, while yielding within blocking calls to respect the progress semantics.

3.2. Improving Threading Support in MPI

Tackling challenges related to the interoperability between MPI and threads is not a new topic. Researchers have explored the topic from different perspectives, ranging from algorithmic optimizations for specific aspects of the runtime to synchronization granularity optimizations and standard extensions.

Algorithmic optimizations target specific components of an MPI implementation. For instance, Gropp and Thakur provided an efficient algorithm for generating context ids in a multithreaded context [Gropp and Thakur 2007]. Goodell et al. showed that concurrent accesses from multiple threads to MPI objects can be a bottleneck when using reference counts, and they proposed more scalable solutions [Goodell et al. 2010].

Granularity optimizations, on the other hand, have been proposed to reduce the extent of thread synchronization. Gropp and Thakur presented an exhaustive thread-safety requirement analysis of MPI functions and their implementation issues [Gropp and Thakur 2007] Their study showed that not every routine requires locking. Most production MPI implementations ensure thread safety of the core functionalities through a coarse-grained critical section. The exception is MPICH, which exploits fine-grained locking on IBM Blue Gene systems. We investigated this fine-grained design in our prior work, where we compared different levels of critical-section granularity and their implications in terms of performance and implementation complexity [Balaji et al. 2010; Dózsa et al. 2010].

Over time, the MPI standard has been extended to support more efficient multithreaded MPI implementations. Hoefler et al. showed that `MPI_Probe` cannot be used by multiple application threads at the same time and that a conventional lock-based workaround is not scalable [Hoefler et al. 2010]; they proposed an efficient solution that goes beyond the implementation level and requires changing the MPI standard. In order to ensure contention-free multithreaded communication, close to that of multiprocess-driven communication, the concept of MPI endpoints has emerged as a potential extension that is being reviewed for inclusion in the next version of the MPI standard [Dinan et al. 2013].

Existing works, however, have ignored the arbitration and hand-off aspects of locking in MPI implementations. To the best of our knowledge, we were the first to tackle this challenge from an arbitration perspective in our prior publication [Amer et al. 2015c]. The current paper extends our prior work by providing substantially more analysis and bringing more insight into the interoperation between locking and MPI runtime progress. We also provide a complementary analysis of the trade-offs between arbitration and latency hand-off in locking implementations. Furthermore, we include results with a hybrid MPI+OpenMP particle transport proxy application that showed tremendous gains from our improved MPI implementation.

3.3. Thread Safety in MPICH

In this paper, we use MPICH on commodity InfiniBand clusters that uses a single coarse-grained critical section[1]. Figure 1 shows a simplified diagram of such a threaded MPI design, with two main execution paths: (1) the main path that all calls are taking, which most of the time yields useful work; and (2) a progress loop used by blocking calls in order to wait for the completion of a blocking operation. We note that in order to respect MPI's progress semantics, the critical section in the progress loop must yield to allow other threads to make progress.

In addition, MPICH has only a single communication context, or endpoint, per process. As a result, the message rate is bound by the the performance of the lower messaging layer, since multiple threads are contending for the same communication context. Figure 2 shows multithreaded message rate results with both the low-level InfiniBand Verbs (`ibverbs`) interface and MPICH (which depends on the former) between two Haswell nodes. In both cases communication performance degrades with the number of threads with small and medium message sizes. We note that message rates are bound by the single-threaded performance when using a single `ibverbs` communication context. With single byte messages, we also observe that the MPICH single-threaded message rate matches closely that of `ibverbs` (roughly 4M messages per second) with a slight added overhead since MPICH is higher in the software stack. More important, however, at full node concurrency (36 cores), the MPICH message rate is almost 4x below that of `ibverbs` (roughly 1M messages per second). This implies that MPICH is adding several times more thread-safety overhead.

Another important detail about the critical section in MPICH is that it uses the system-provided Pthread mutex. This is a common practice for many thread-safe software packages, including other MPI implementations. In particular, because our platforms are Linux-based clusters, the Pthread mutex API is provided by the Native POSIX Threading Library (NPTL) [Molnar 2003]. In the remainder of the paper, we will refer to the NPTL implementation of the mutex lock as the "Pthread mutex." The Pthread mutex is blocking; in other words, the calling thread might block in the kernel if it fails to acquire the lock. The Pthread mutex has two components. First, in the user-space, a thread tries to acquire the lock by using hardware atomics, usually with a compare-and-swap (CAS) operation. Second, in the kernel-space, if the acquisition fails, the thread goes to sleep in the kernel using

[1]This is true for a performance-oriented build of MPICH 3.2. Additional locks, such as those that protect memory allocation, are used for other type of builds, such as for debugging purposes.

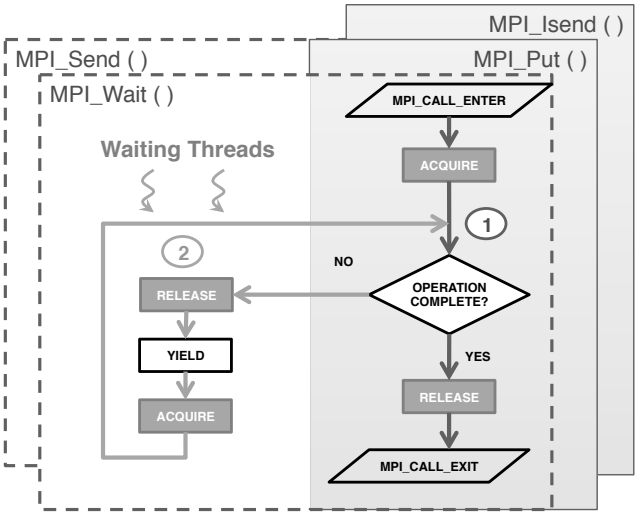
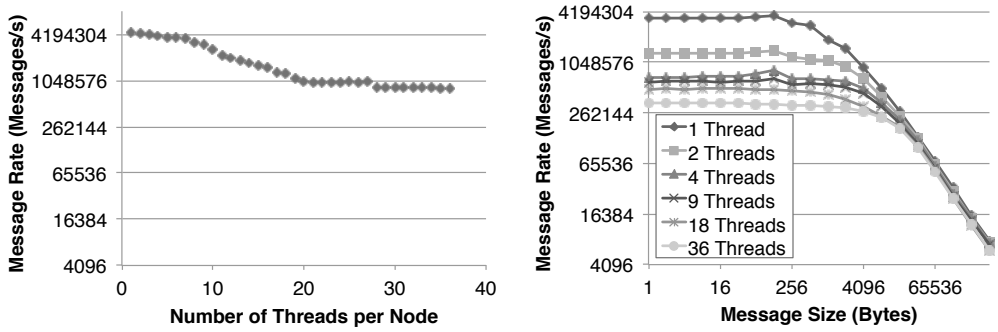


Fig. 1. Characteristics of the thread-safety implementation in MPICH: simplified diagram of a coarse-grained critical section design.



(a) Single-byte multithreaded message rate with `ibverbs` with a single context

(b) Multithreaded message rate with MPICH

Fig. 2. Message rate between two Haswell nodes using a single communication context with (a) the low-level `ibverbs` messaging layer and (b) MPICH.

the `FUTEX_WAIT` operation of the `futex` (fast user-space mutex) system call [Franke et al. 2002; Drepper 2005]. When the lock holder leaves the critical section, it wakes up blocked threads, usually at most one, with the `FUTEX_WAKE` operation. The threads awake and return to the user-space and compete again for the lock using hardware atomics. Although some variations might exist across hardware, Linux kernels, and NPTL versions, the above two-phase implementation holds for most platforms. In the next section, we will analyze the behavior of the Pthread mutex and other popular locks in terms of arbitration and hand-off latency.

4. ARBITRATION AND HAND-OFF LATENCY ASPECTS

This section presents an overview of some widely used locks and discusses their arbitration and hand-off latency properties.

4.1. Analysis of the Pthread Mutex Fairness

Test-and-set type of locks can experience starvation issues. The main cause is a combination of the competitive nature of the hand-off, unlike the explicit direct hand-off of FIFO locks, and the nonuniform cache access hardware, which biases the arbitration according to cache proximity. The user-space component of the Pthread mutex follows a similar model. How the kernel component affects the arbitration, however, is less understood. Let us assume that the lock holder will try to request again the lock immediately after releasing it. For simplicity, we refer to a thread performing a `FUTEX_WAKE` operation the transmitter of a signal, and the thread that gets woken up the receptor of the signal. Depending on the time the transmitter takes to issue the signal and to return to the user space and the time the receptor wakes up and also returns to the user space, we distinguish two main possibilities: (1) if either thread returns to the user space before the other, then the first thread will have a higher chance of acquiring the lock if no other competing thread is in the user space; and (2) if both threads return to the user space at roughly the same time, the transmitter will be favored because of cache proximity.

To get insight into the latency of the futex signal-wakeup mechanism from the transmitter and receptor perspectives, we developed a microbenchmark that estimates those latencies. Since `FUTEX_WAKE` is nonblocking, measuring the latency of issuing the signal and returning to the user-space is straightforward. Estimating the time it takes to wake up from `FUTEX_WAIT` is harder, however, because it requires kernel-level instrumentation to extract the signal arrival time. Instead, we adopted a conservative approach where we assume that signal latencies are shorter and deduce an underestimated wakeup latency. We performed a ping-pong synchronization pattern between a pair of threads while measuring the total time and the local signal time at each thread, as shown in Figure 3. We derived the total wakeup time by subtracting the signal times from the total. We clearly underestimated the wakeup time because we discarded the period between a receptor receiving the signal and the transmitter reaching the user space. We also overestimated the signal cost because we included the busy waiting time to make sure the receptor thread was successfully blocked in the kernel space.

Our latency estimations on a Haswell node are shown in Figures 4a and 4b. The signal latency results are shown as a matrix of latencies between every pair of cores on the node. The wakeup latencies are shown as a similar matrix but as a factor of the signal latency. First, we observe that the latencies are biased by the memory hierarchy. We also note that the wakeup delay is longer than the signal latency despite our underestimations (1.2x to 1.7x times longer). This result supports our hypothesis that signal latencies are shorter than wakeup latencies. As a result, the kernel component also biases the arbitration in favor of the last lock holder.

Another important aspect of the Pthread mutex implementation is the detection of waiting threads. A thread releasing the lock issues wakeup signals only when it sees a user-space flag was set (by waiters before blocking in the kernel). This flag is cleared by the transmitter when issuing a signal, and another waiter has to reset it again. Because of the longer wakeup delays, a thread can monopolize the lock before a waiting thread resets the flag. As a result, `FUTEX_WAKE` system call overheads are amortized by the lock monopolization, and the overall hand-off latency is expected to be low. Furthermore, we confirm that the current Pthread mutex implementation is unfair and heavily biased.

Starvation can be a serious issue for an MPI implementation since it cannot fulfill the MPI progress semantics and lead to deadlocks. This can be illustrated with a scenario where a thread within each MPI process monopolizes the lock while waiting to receive a message. These receive operations are not serviced since threads issuing the corresponding send operations are blocked by the receiving threads. In practice, however, threads can leave the starvation state in Pthread mutex. Although the arbitration is biased in favor

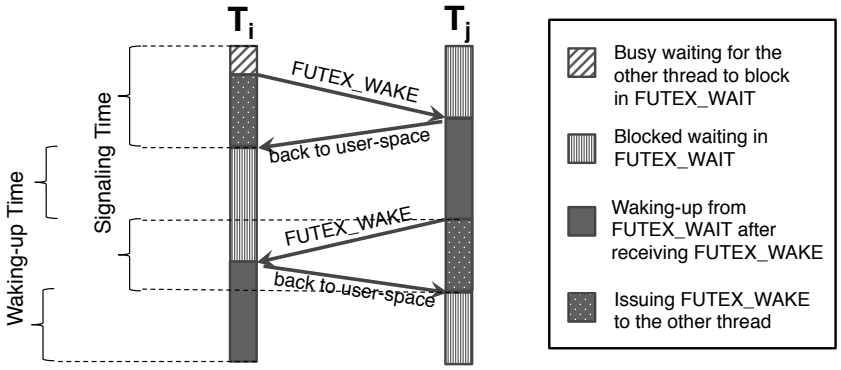
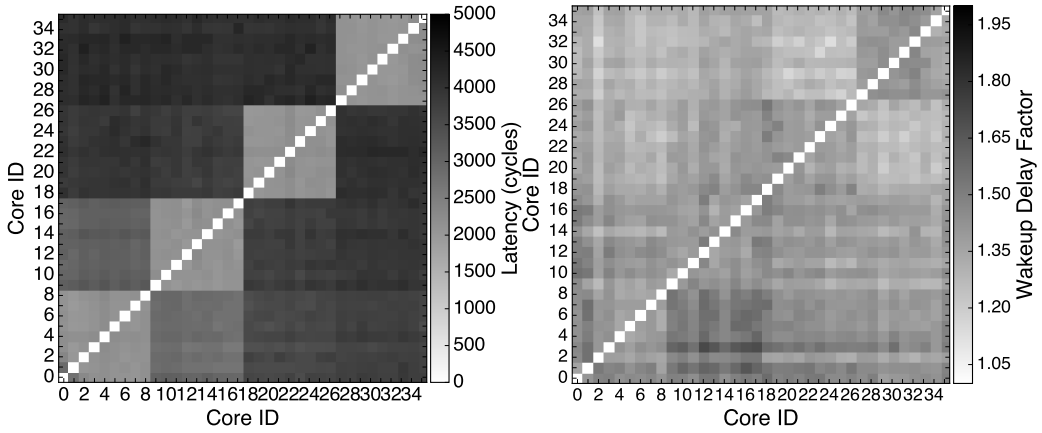


Fig. 3. Ping-pong benchmark with the `futex` system call.



(a) Signal latency (overestimate) (b) Wakeup delay factor

Fig. 4. Time for a thread to execute `FUTEX_WAKE` (signal time) and delay of the receiver thread to wake up and return to the user space. The wakeup delay is shown as a factor of the signal time.

of the previous lock holder, it does not guarantee it acquisition. For instance, a thread could grab the lock while the previous lock holder is calling `FUTEX_WAKE`. We will show in Section 5 how this arbitration affects communication progress. First, however, we will discuss the trade-off between fairness and hand-off latency. This discussion will also serve as an introduction to the basic locks used in the later sections.

4.2. Fair Arbitration and Hand-off Latency

The advantages of the Pthread mutex are its ease of use (users do not need to implement a lock themselves and only need to use an API); its portability across many platforms, including Linux and BSD systems; and its usual efficiently in uncontested cases[2]. It is in contested cases that the Pthread mutex is usually criticized. If the arbitration issues are harmless—for example, any thread can make progress inside the critical section, or the mutex does not cause significant load-balancing issues—the throughput of the Pthread mutex is competitive with that of other locks thanks to its lock monopolization, which

[2]CAS on an uncontested address is cheap on many platforms, and transactional memory is being explored to improve the fast path of Pthread mutex in NPTL.

improves the locality of reference of the lock and the critical section data and thus the throughput of the lock. In addition, because a subset of the threads is in the kernel space, this acts as a back-off mechanism that reduces the cache coherency traffic and thus improves the hand-off latency as well.

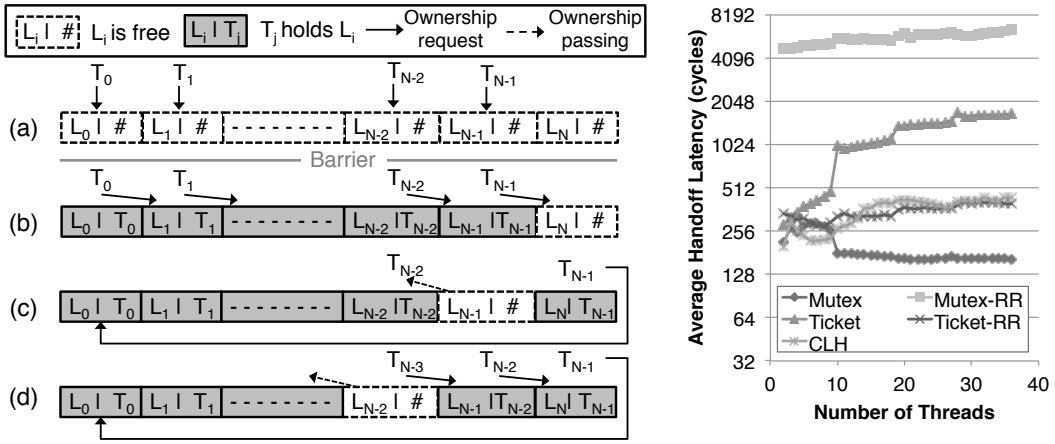
We compared the hand-off cost of the Pthread mutex with two popular lock implementations—the ticket lock [Mellor-Crummey and Scott 1991a], and the CLH queue lock [Craig 1994]—in two settings. First, we measured the hand-off latency when protecting an empty critical section with a single lock[3]. Second, we measured the *direct* hand-off cost in a round-robin lock-passing pattern. For this purpose, we developed a benchmark that uses an array of locks of length equal to the number of threads in addition to one spare lock, as illustrated in Figure 5a. Each thread acquires a lock in the array (step (a)) and then attempts to acquire the next lock in the array (step (b)) before releasing the previously owned lock. This strategy ensures that no deadlocks occur and that at most one lock acquisition occurs at each timestamp thanks to the spare lock. The spare lock moves like a bubble in the array (steps (b–d)). This benchmark allows us to measure easily the hand-off latency of any lock implementation while avoiding misleading self hand-offs that occur with lock monopolization.

We remind the reader that the ticket and CLH locks ensure FIFO arbitration. The ticket lock uses two shared counters. The first counter is incremented atomically by each thread to get a unique ticket and wait its turn, and the second counter is incremented at release time to grant critical section access to the next thread. On cache-coherent systems, the ticket lock is known to suffer from the cache-coherency traffic because of the shared-global state of the lock. CLH is a queue lock that uses pointer-swapping operations to build a FIFO queue during lock passing. Here, sharing occurs between at most two threads, which reduces the sensitivity to the cache coherency issues of the ticket lock.

Figure 5b shows the average hand-off latency with respect to the number of threads in the system on the Haswell machine. `Mutex` and `Ticket` denote the normal use of the locks with the empty critical section, while `Mutex-RR` and `Ticket-RR` are the same locks but used in the round-robin lock-passing benchmark. We observe that the normal use of the Pthread mutex enjoys the lowest hand-off latency. The ticket and CLH locks are sensitive to the memory hierarchy, with the ticket lock suffering more because of its all-shared state. The Pthread mutex alleviates both issues in this case thanks to its lock monopolization, which ensures low-latency self and neighbor thread hand-offs. When forcing round-robin lock passing, however, we observe that the round-robin hand-off latency of the Pthread mutex is much higher. We also notice that the hand-off latency is on the same order of magnitude as that of the futex signal and wakeup latencies (cf. Figure 4). This round-robin lock passing forces costly kernel-level wakeups on the critical path. Unlike in the default Pthread mutex behavior, the costs of these wakeups are not amortized by user-space lock monopolization. In addition, we notice that `Ticket-RR` performs similarly to the CLH lock and better than the normal usage of the ticket lock. The reason is that in the round-robin benchmark, synchronization occurs only between two threads and thus reduces the cache coherency overhead similar to the way the CLH lock does. In fact, any user-space lock that does only a single atomic operation will behave similarly to the CLH lock in this benchmark.

We conclude that having control over the arbitration also requires taking into account the hand-off latency. If the lock implementation involves blocking in the kernel, the cost of the kernel signal/wakeup mechanisms needs to be properly amortized; otherwise it will outweigh the benefits of a better arbitration. In the following sections, the fair locks and our customized lock are purely in the user space to benefit from low-latency hand-offs.

[3]Note that no consensus exists on the right way to benchmark lock implementations. Consequently, the best-performing lock might vary according to the time spent and the size of the data accessed inside and outside the critical section



(a) Round-robin lock-passing benchmark. (b) Lock hand-off latency

Fig. 5. Normal and round-robin hand-off latency of various lock implementations.

5. EFFECT OF LOCK MONOPOLIZATION ON COMMUNICATION PROGRESS

We showed in the preceding section that the Pthread mutex can potentially induce lock monopolization in contested cases. Whether this currently occurs in an MPI implementation and whether it affects performance in any way remain unknown. In this section, we proceed with our analysis in three steps: (1) prove the existence and extent of the lock monopolization; (2) show how it incurs wasteful lock acquisitions; and (3) correlate the lock acquisitions to the internals of the MPI runtime and communication progress.

5.1. Runtime Contention and Lock Monopolization

To assess the degree of monopolization in a lock acquisition sequence $S = \{A_1, A_2, ..., A_X\}$, we provide several definitions and then derive a metric to be used in our empirical analysis.

Definition 5.1 (*Fair Lock Acquisition*). Given a set of N threads waiting to enter a critical section, if the arbitration is fair, any waiting thread has a probability $\frac{1}{N}$ of being the next lock owner.

Note that this definition of fair acquisition is more relaxed than that of the strict FIFO policy because we allow a thread to reacquire the lock despite the existence of waiting threads as long as the reacquisition is done with a fair probability. In the following we give two definitions that help us build our fairness metric.

Definition 5.2 (*Monopolization Score*). The monopolization score of a given lock acquisition sequence S is defined as $M(S) = \sum_{i}^{X} A_i$, where $A_i = 1$ iff acquisition A_i granted the lock to the same thread as A_{i-1} while more than one thread was waiting; $A_i = 0$ otherwise.

Definition 5.3 (*Fair Score*). The fair score of a given lock acquisition sequence S is defined as $F(S) = \sum_{i}^{X} P(A_i|N)$, where $P(A_i|N)$ is the probability of fair lock acquisition knowing N threads are waiting.

Given these score definitions, we can quantify the degree of bias of a lock for a lock acquisition sequence S by comparing the monopolization score with the fair score.

Definition 5.4 (*Bias Factor*). The bias factor of a given lock acquisition sequence S is defined as $B(S) = \frac{M(S)}{F(S)}$.

This factor can be defined for any arbitrary level of the memory hierarchy. The definitions above assume a thread-level or core-level granularity. By changing the scores to reflect clustered threads in a particular level of the memory hierarchy (e.g., NUMA-node level), we can derive the bias factor at that level as well. Intuitively, a high bias factor reflects severe lock monopolization. Bias factors close to 1 imply a close-to-fair arbitration during the acquisition sequence. Bias factors below 1 are possible when the same thread rarely gets the chance to successively reacquire the lock while waiting threads exist, such as in the case of FIFO locks. We say in this case that FIFO has a strong fairness requirement.

We augmented MPICH with a sampling-based tracing infrastructure to record lock acquisition sequences and derive the bias factors at the core and NUMA-node levels. Specifically, we associated a waiting thread counter with each NUMA node. Threads update the counter of their corresponding NUMA node and then compete for the lock. The winner records the statistics after acquiring the lock[4]. We measured the bias factor when using the baseline Pthread mutex lock for a multithreaded message-rate benchmark. Figure 6a shows the results. We observe bias factors greater than one at the core level, and at the NUMA-node level when crossing the second socket, which indicate lock monopolization is truly taking place within the MPICH runtime. In addition, this metric indicates the degree of contention to some extent since waiting threads are taken into account. We also observe that the monopolization issue magnifies with higher degrees of thread concurrency. In the next two subsections, we correlate this observation with communication progress in MPICH.

5.2. Analysis of Wasted Progress

Polling lower network layers for communication progress is an important part of an MPI implementation. Although completion can be signaled by using interrupts, active polling is often faster at detecting the progress of a communication operation. Polling, however, can be expensive,[5] and wasteful if no progress has been made and other work could have been executed in the meantime. In fact, in several messaging interfaces, including the MXM interface used in this study, a single poll forwards the completion of all outstanding operations; and as a result most subsequent immediate polls do not usually yield any progress.

From the simple coarse-grained critical section design of MPICH in Figure 1, we observe that threads can block waiting in the progress loop. When an outstanding operation is not complete, the usual model is to poll the lower messaging layer and check again whether the operation has completed. Since we know that lock monopolization occurs within the runtime, waiting threads may be monopolizing the lock in the progress loop and incurring unnecessary extra polls. In the following, we define two types of unnecessary polls that will help us derive our next metric.

Definition 5.5 (*Unsuccesful Progress Poll*). An unsuccessful poll is a polling operation that did not yield any progress.

Definition 5.6 (*Wasted Progress Polls*). A wasted progress poll is an unsuccessful polling operation that occurred while other threads are waiting to enter the main execution path.

The idea behind the wasted progress poll is that instead of polling, other threads could have made use of the lock acquisition by performing work, such as issuing operations or freeing resources. We instrument MPICH to record wasted polls and derive their ratio among

[4]Although this instrumentation adds overhead inside and outside the critical section, our empirical results showed overheads below 10%.
[5]For example, we measured hundreds of nanosecond per poll with recent versions of MXM.

unsuccessful polls. We analyze the message rate benchmark and show the results in Figure 6b. We observe that a large percentage of the unsuccessful polls are wasteful. In the next subsection, we show some types of work that could have been performed instead of polling unnecessarily.

5.3. Analysis of Message Requests

In this section, we illustrate how detecting the completion of an outstanding operation can be delayed in the context of point-to-point communication.

The message rate used throughout this analysis is derived from the single-threaded bandwidth benchmark of the OSU benchmark suite[6]. In short, on the sender side, each thread issues a window of `MPI_Isend` operations and waits for them in a batch with an `MPI_Waitall` blocking call. Conversely, threads on the receiver side issue windows of `MPI_Irecv` operations and wait for them. In order for the user threads to detect that all operations have completed, a thread in `MPI_Waitall` has to acquire the lock to verify that. As a result, a delay could occur between when the operations have effectively completed and the time when the user threads detect them. For good performance, this delay has to be as short as possible in order to reduce latency and improve communication throughput. To help assess the efficiency of the runtime in this regard, we tracked completed and freed requests, namely, those detected by the user thread; and we derived the following metric.

Definition 5.7 (*Dangling Requests*). A dangling request is a request that has completed but has not yet been detected and freed by the user.

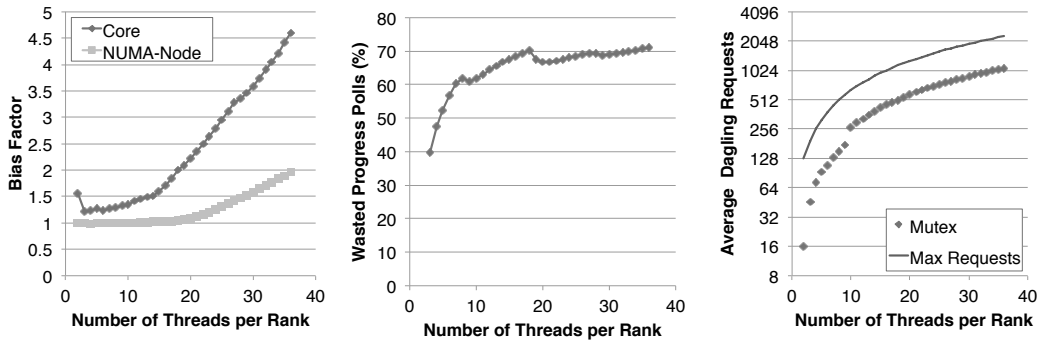
We extended MPICH to instrument request-related information through the `MPI_T` interface, and we combined it with a sampling-based tracing approach to record completed and freed requests. We derived from the traces the average number of dangling requests during execution. Figure 6c shows the dangling requests analysis for the message rate benchmark [7]. We show results on the receiver side; the sender side follows a similar trend. We notice that the dangling requests grow with the degree of thread concurrency and reach more than a thousand on average with 36 threads. With window sizes of 64 requests per thread, roughly half the maximum allocated requests are dangling. Completion detection is important because it is on the critical path and can affect issuing subsequent communication operations. Specifically, slow completion detection negatively affects communication latency and throughput.

5.4. Reception Rate and Message Queues

One type of critical objects that MPI implementations rely on are message queues. We evaluate in this section the rate at which an MPI process is receiving messages while monitoring the state of the posted receive queue. This queue is used for message matching where a receive request that cannot yet be satisfied with a matching send is pushed to the posted queue. This queue reflects the readiness of the receiver to consume messages. Posting receive operations in advance is favored over letting unexpected messages accumulate at the receiver. Involving the unexpected queue in the critical path can occur when a receiver cannot keep up when flooded with messages. Its use has several shortcomings, including additional memory copies, because reception buffers are unknown, and more expensive control flow when the unexpected queue is saturated. Thus, readiness to receive is generally a good property, although exceptions exist, such as when the posted queue is long and in-

[6]http://mvapich.cse.ohio-state.edu/benchmarks

[7]Our sampling-based tracing incurs low overheads here and the subsequent analyses as well. Due to the batch nature of our benchmarks, however, variations around the average in our measurements can be high. We do not show standard deviations for clarity reasons, though be believe the average numbers reported do not lose in accuracy and insight



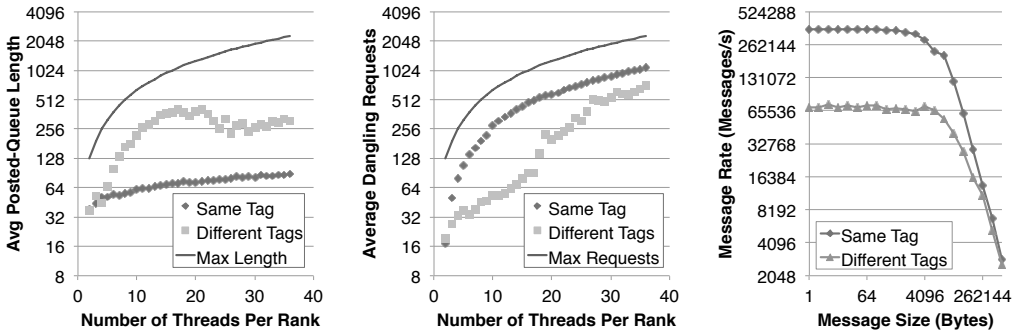
(a) Arbitration fairness (b) Wasted progress (c) Dangling requests

Fig. 6. Analysis of unfair arbitration and its consequence on communication progress in the point-to-point throughput benchmark from the receiver's perspective. Windows of 64 requests per thread were used.

curs non-negligible matching overheads and prohibitive memory consumption. The previous message-rate benchmark is symmetric with respect to threading, because both sender and receiver are multithreaded. As a consequence of the locking overheads, both processes are slowed. To get better insight into the reception capabilities, we used a different benchmark that relies on a single-threaded contention-free sender and a multithreaded receiver. We explored two variants:

Same Tag. All message tags are the same, and thus threads at the receiver are capable of consuming any message.
Different Tags. Threads consume messages with exclusive tags. This process simulates a thread-to-thread communication pattern, where the sender is communicating to a specific thread by identifying it with a unique tag. In addition, the sender does not issue the next window of messages until all messages to all threads in the current window are consumed.

In both cases, we expected the issuing rate of receive operations to be slowed because of the lock monopolization. The same-tag case is less sensitive to this shortcoming because threads monopolizing the lock are able to post new windows and help the receiver keep up with the sender to some extent. In the different-tags case, on the other hand, monopolizing threads can issue at most one more window and wait for long periods because the sender will not issue the next window until all receiving threads have consumed all the messages of the current window. Although the active set of monopolizing threads changes over time, the change is slow, and consequently the reception rate is slow as well. We observe in Figure 7a that the length of the posted-queue in the same-tag case is less than the equivalent of two windows of receive operations. At full concurrency, it is 25× lower than the maximum number possible of outstanding receive operations. On the other hand, the completion of requests that get pulled from the posted queue is not necessarily detected by the corresponding thread and consequently increases the number of dangling requests (Figure 7b). When using different-tags, the number of posted receive operations increases because monopolizing threads cannot make progress for long periods; the active set of monopolizing threads changes, and other threads post new operations. Dangling requests also increase considerably but are fewer than in the same-tag case. Similarly, threads have slightly more chances to free requests in the different-tag case because the change in the active set of monopolizing threads allows freeing requests, which translates into less dangling requests. When looking at raw performance in Figure 7c, however, we can see that the throughput with different-tags drops significantly. Here, lock monopolizers are mostly wasting progress



(a) Posted-queue length (b) Dangling requests (c) Message-Rate

Fig. 7. Single-sender multithreaded receiver comparison between the same-tag case, when any thread can consume the messages, and the different-tags case, when messages are targeted to specific threads: (a) analysis of the posted receive message-queue and (b) message rate with 36 threads/rank. The results were obtained with window sizes of 64 receive operations per thread at the receiver and $64 \times Nthreads$ send operations at the sender, where $Nthreads$ is the number of threads at the receiver.

polls, since the sender is waiting to issue the next batch, while not allowing other threads to post new requests or free their dangling requests. Consequently, the readiness of the receiver is severely hindered and translates into a large drop in throughput.

In the next section, we show how different locking implementations can alleviate these issues and, in fact, bias the arbitration in favor of making faster progress rather than slowing it.

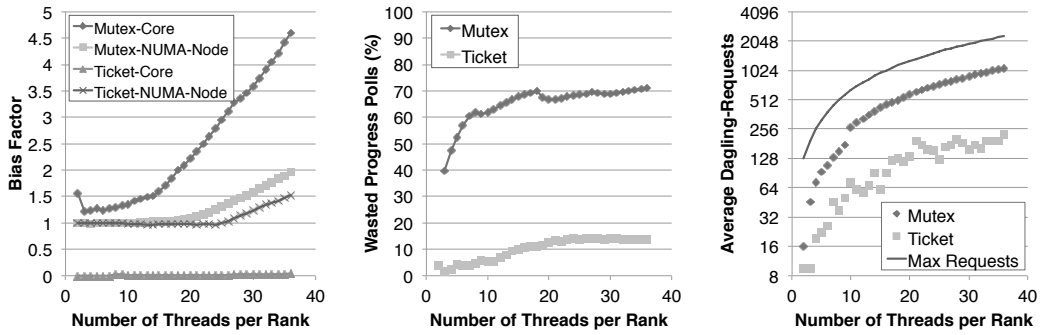
6. ALTERNATIVE LOCKING FOR BETTER PROGRESS

The preceding section showed a correlation between lock monopolization, induced by the Pthread mutex, and inefficient communication progress. In this section we explore alternative locking methods to alleviate those issues. Furthermore, Section 4 showed that altering the arbitration policy while keeping the kernel component requires careful amortization with user-space lock passing. Instead, we explore pure user-space locking options, which provide more flexibility at relatively lower hand-off latencies.

6.1. First-In, First-Out Arbitration

As our first option, we analyzed the effect of introducing fairness to the lock acquisition and break the lock monopolization cycles. Several locks exhibit the FIFO arbitration property, including the ticket lock [Mellor-Crummey and Scott 1991a] and queue locks such as MCS [Mellor-Crummey and Scott 1991b] and CLH [Craig 1994]. In this section we use the ticket lock mostly for analysis purposes; in the evaluation section we present results with the CLH lock as well.

Using the same message-rate benchmark described in Section 5.3, we performed a comparative analysis with the Pthread mutex in terms of fairness and effect on communication progress. Figure 8a shows that the bias factor of the ticket lock at the core level is close to zero, which is due to its strong FIFO restrictive arbitration. At the NUMA-node level, however, the ticket lock is slightly biased because it is faster for threads on the same NUMA node to increment the shared counter. Looking at the wasted process polls in Figure 8b, we see that the ticket lock keeps wasteful progress much lower than does the Pthread mutex. The remaining wasted polls with the ticket lock are due to FIFO passing, where consecutive acquisitions can cause unnecessary polls while other threads are blocked at the main path entry. We also compared dangling requests and show the results in Figure 8c. We notice



(a) Arbitration Fairness (b) Wasted progress (c) Dangling requests

Fig. 8. Comparative analysis of the Pthread mutex and ticket lock in terms of fairness and effect on MPI communication progress.

that the average number of dangling requests is much lower than when using the Pthread mutex, and also the growth rate is slower.[8] Although not shown in the graph, the number of dangling requests varies significantly during execution. In particular, the ticket lock incurs more variability than when using the Pthread mutex. This is due to the nature of the benchmark. Since the communication is performed in batches, requests get completed and freed in batches as well, translating into large variations in our profiling. The Pthread mutex has less variation because a small set of threads monopolizes the lock and the dangling requests of the other threads remain inside the runtime longer, resulting in more stable numbers.

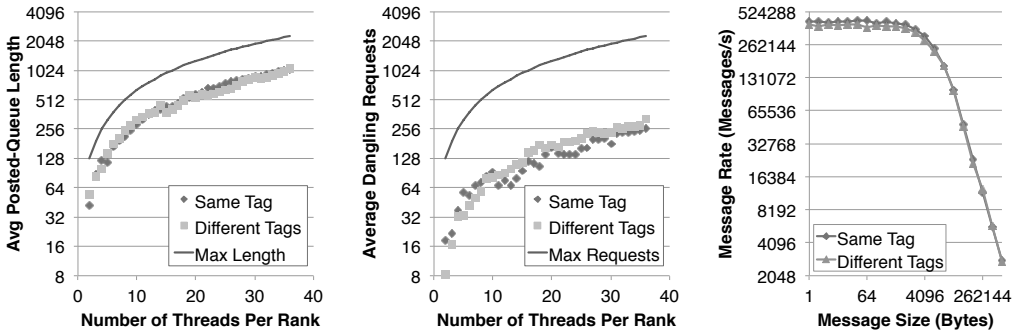
We also compared the reception capabilities of MPICH when using the ticket lock in the single-sender benchmark. In Figure 9a, we observe that the ticket lock allows a high number of posted requests to be maintained regardless of the tagging pattern. In addition, dangling requests are kept low regardless of the tagging patterns (Figure 9b). This result translates into a throughput for the different-tags case close to that of the favorable same-tag pattern, as shown in Figure 9c. Maintaining a low overhead when heterogeneous matching patterns exist among threads is an important property of an MPI implementation. It can allow efficient and flexible communication patterns, such as creating virtual topologies among threads using communicators and tags.

6.2. Obstructing Progress Waiters with Priority Locking

After understanding the effect of the arbitration on the internals of the MPI implementation, we biased the arbitration to favor cases with higher chances of doing useful work, and we obstructed progress waiting threads because they have lower chances of making progress.

It is not trivial to know a priori which thread is going make progress after getting the lock. The situation depends in many cases on external events such as message reception. Nevertheless, we can identify unbalanced execution paths in the runtime that yield different amounts of work within the critical section. Looking back at Figure 1, we recall the existence of two distinct execution paths: (1) a *main path* that each thread-safe routine implements differently and (2) a *progress loop* that some MPI routines enter to poll for communication progress. For instance, `MPI_Irecv` may allocate and enqueue a request in the posted queue on the main path, while not going through the progress loop. On the other hand, a blocking call, such as a `MPI_Wait`, will enter the communication progress engine until the corresponding request completes. We note that threads in the *main path* have more chances to yield useful work and thus are unlikely to waste a lock acquisition.

[8]less than 10% as opposed to roughly 50%, for the Pthread mutex, of the maximum number of requests



(a) Posted-queue length (b) Dangling requests (c) Message rate

Fig. 9. Effect of FIFO locking on message reception. The ticket lock was used as a representative of FIFO locks.

Arguably, however, some MPI operations, such as `MPI_Test`, may not make progress even along the main path. The ticket lock circulates ownership independently from the path a thread takes. This includes cases where threads are blocked at the entry of the main path while waiting for threads in the progress loop to release the lock. Such cases may also yield to wasted lock acquisitions, as shown in Figure 8b.

Given these observations, we decided to obstruct progress waiters in favor of threads on the main path. To this end, we used a two-level priority lock. Our lock implementation resembles cohort locks or hierarchical locks [Chabbi et al. 2015], where threads sharing some level of the memory hierarchy perform local passing before letting another group of threads acquire the lock on a different domain of the memory hierarchy. That is, we used a two-level lock where threads that belong to the same class acquire their local lock first, before the global one. The main difference from cohort locks is that we do local passing only between high-priority threads; low-priority threads have to acquire both locks every time. Figure 10 illustrates how a two-level CLH-ticket lock is implemented. We exposed two API routines for the two classes of threads. The regular interfaces are for normal usage that we consider as high priority. The low-priority interface (prefixed with `_low`) is meant for low-priority execution paths. The data structure of this lock uses two CLH locks, `high` and `low`, corresponding to the two classes of threads. We also added a flag `go_straight` to be set by the first high-priority thread to inform subsequent high-priority threads to not acquire the second lock. The second lock, `filter`, is a ticket lock that acts as a wall to block low-priority threads when high-priority threads are present. In order to check whether high-priority threads are waiting, an additional field, `next`, was added to the original CLH `qnode`. This field has the same meaning as the `next` field in an MCS lock, but it does not require synchronization between the lock holder and the next owner, and consequently conserves the zero atomic operation property of the CLH lock at release time.

This lock was designed to add little overhead when prioritizing the main path does not improve efficiency, where circulating the lock in FIFO fashion is sufficient. The lock is helpful when a large number of operations need to be issued, such as a large window of two-sided communication. Indeed, in the message rate benchmark the benefit of the lock grows with the size of the window, as will be shown in the next section.

In the previous message rate benchmarks, delaying a thread to issue an operation can be amortized by another thread issuing the same type of operation, since threads within the same rank are whether all receiving or everyone is sending. To be able to stress operation issuing further, we implemented a microbenchmark with an all-to-all communication pattern. This benchmark is similar to the previous message rate test with the exception that

```
/* Node Data-Structure */
typedef struct clhp_qnode {
    bool waiting;          // waiting flag
    clhp_qnode* prev;      // keep track of the previous lock holder's qnode
    clhp_qnode* next;      // check if a thread is queued behind the lock holder
} clhp_qnode;

/* Lock Data-Structure*/
typedef struct clhp_lock {
    clhp_qnode high;      // FIFO between high-priority threads
    bool go_straight;     // allow lockless second phase to high-priority threads
    ticket_lock filter;   // filter lock to block low-priority threads
    clhp_qnode low;       // FIFO between low-priority threads
}

/* Regular/High-Priority Interfaces */
void clhp_acquire(clhp_lock *L, clhp_qnode* I) {
    I->waiting = true;
    I->prev = NULL;
    /* acquire the high-priority CLH lock */
    clhp_qnode* pred = I->prev = atomic_swap(L->high, I); // swap pointers
    pred->next = I;             // inform prev high-priority that I am waiting
    while (pred->waiting);      // spin
    if (!L->go_straight) {      // if I am the first high-priority thread
        ticket_acquire(filter); // acquire the filter lock
        L->straight = 1;        // inform next priority threads to go straight
    }
}
void clhp_release(clhp_lock L, clhp_qnode* I) {
    if(I->next == NULL) {        // no high-priority thread is waiting
        L->go_straight = 0;      // force high-priority threads to acquire filter
        ticket_release(L->filter);// allow low-priority threads to pass
    }
    /* release the high-priority CLH lock */
    clhp_qnode* pred = (*I)->prev;
    (*I)->waiting = false;
    *I = pred;                  // reuse pred's qnode
}

/* Low-Priority Interfaces */
void clhp_acquire_low(clhp_lock *L, clhp_qnode* I) {
    I->waiting = true;
    I->prev = NULL;
    /* acquire the low-priority CLH lock */
    clhp_qnode* pred = I->prev = swap(L->low, I); //swap pointers
    while (pred->waiting);      // spin
    ticket_acquire(filter);     // unconditionally acquire the filter lock
}
void clh_release_low(clhp_lock L, clhp_qnode* I) {
    ticket_release(L->filter);  // unconditional release the filter lock
    /* release the low-priority CLH lock */
    clhp_qnode* pred = (*I)->prev;
    (*I)->waiting = false;
    *I = pred;                  // take pred's qnode
}
```

Fig. 10. Priority locking pseudo-algorithm with two levels of priority.

threads here are both sending and receiving from all processes. This is achieved by issuing a window of nonblocking send and receive operations and waiting for them. Here, issuing is more critical to keep a good balance between posting receive operations and sending messages.

First we analyze the behaviour of the CLH lock and the priority lock (CLH-LPW)[9] introduced before in the context of the all-to-all benchmark. In Figure 11a, we observe that the CLH-LPW lock reduces, and close to eliminating, wasted progress polls, which is an expected outcome. Another expectation is that dangling request may increase, because most of the request completion is detected at the progress engine level and waiting threads are obstructed. Figure 11b confirms this hypothesis, where we observe that dangling requests increase almost two-folds. When looking at the length of the posted receive queue in Figure 11c, the results are somewhat surprising since giving priority to issuing requests should have raised the length of the posted receive queue. The explanation is that issuing faster receive and send operations does not allow for requests to accumulate in the posted receive queue. We measured the issue rate of both send and receive operations and show the results in Figures 11d and 11e. We observe that the issuing rate of the receive and send operations is roughly twice higher in the CLH-LPW lock case. Consequently, with a higher issuing rate and a relatively high receiving readiness, the CLH-LPW lock outperforms the CLH lock by 50%-100% for small and medium message sizes as shown in Figure 11f.

7. EVALUATION

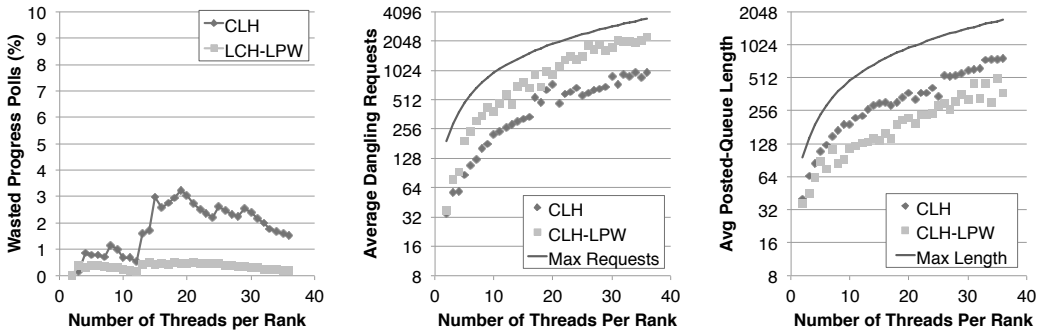
In this section, we present performance results with the different locking implementations using microbenchmarks, stencil and graph kernels, a particle transport proxy application, and a genome assembly application. The point-to-point evaluations were run on the Haswell machine, and the kernels and applications were run on the Nehalem machine.

7.1. Microbenchmarks

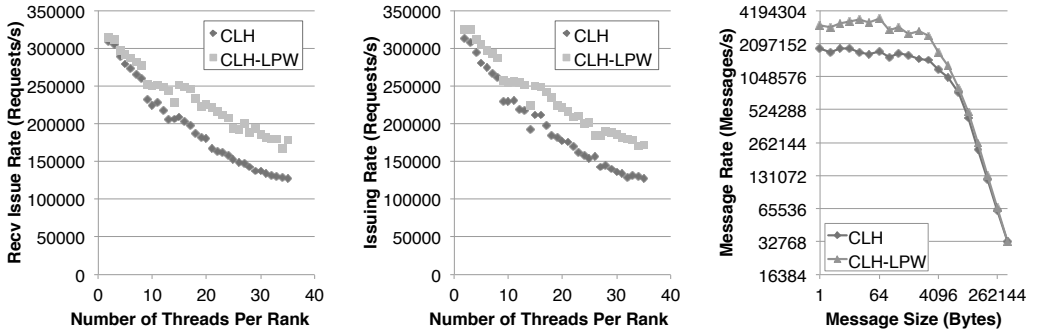
We begin by presenting results with multithreaded two-sided point-to-point throughput and latency benchmarks. We also evaluate MPI one-sided operations with asynchronous progress.

7.1.1. Two-Sided Communication. Figure 12 summarizes the results with throughput and latency benchmarks of all methods using all 36 cores of the Haswell nodes. The latency benchmark was derived from `osu_latency` of the OSU microbenchmarks suite. We modified the benchmark so that both processes are multithreaded, where each thread alternates between issuing blocking send and receive operations. As hinted in the previous section, the CLH-LPW lock performs better than the flat CLH lock with large windows of requests, thanks to prioritizing issuing, which becomes more beneficial. We can confirm this observation by comparing the results with the default window size of 64 (Figure 12a) to a larger window size of 512 (Figure 12b). We also observe that the throughput with the large window gradually improves when going from the Pthread mutex to the ticket, CLH, and CLH-LPW locks, thereby indicating that both arbitration and hand-off latency contribute to this improvement. CLH-LPW achieves the best throughput, outperforming the Pthread mutex by 57% for small and medium messages. Multithreaded latency follows a similar trend (Figure 12c) except that CLH-LPW does not improve efficiency but rather adds a slight overhead. The best-performing lock is CLH, which reduces the latency achieved by the Pthread mutex by up to 137%. CLH-LPW adds an average overhead of 30% for small messages compared with CLH and performs similarly with medium and large messages. Here, since there is

[9]In the context of the MPI implementation, we refer to the CLH-ticket priority lock as CLH-LPW, which stands for CLH with lower priority for waiters.

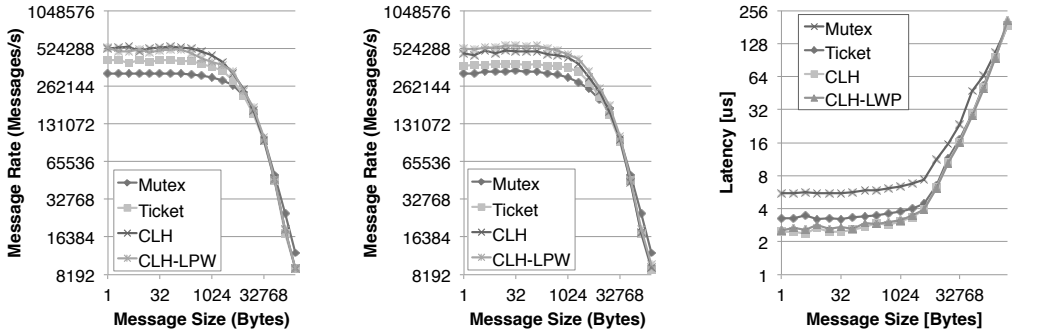


(a) Wasted progress (b) Dangling requests (c) Posted receive queue length



(d) Receive issuing rate (e) Send issuing rate (f) Message rate

Fig. 11. Comparative analysis of FIFO locking and the priority locking in a multithreaded all-to-all communication pattern in terms of issuing and message rates. The data is collected on rank 0 while running with 4 ranks.



(a) Throughput with 64 messages per window (b) Throughput with 512 messages per window (c) Latency

Fig. 12. Performance comparison of the Pthread mutex and ticket, CLH, and CLH-LPW locks with the two-sided point-to-point throughput and latency benchmarks.

only one outstanding request per thread, prioritizing the main path has little effect because most of the time threads are waiting in the progress loop.

7.1.2. Remote Memory Access with Asynchronous Progress. Remote memory access (RMA) provides a powerful model where processes can access memory outside their address space and even outside the physical node they are running on. The Global Arrays [Nieplocha et al. 2006a] library employs RMA to support a user-friendly unified view of distributed arrays, and is used in a number of scientific applications, the most well-known of which is the chemistry package NWChem [Valiev et al. 2010]. The Aggregate Remote Memory Copy Interface (ARMCI) [Nieplocha et al. 2006b] is the abstraction layer for one-sided communication inside of Global Arrays, which is mapped directly to MPI-3 one-sided communication in ARMCI-MPI [Dinan et al. 2012].

We conducted an experiment in which one process does RMA operations (put, get, and accumulate) to or from other processes on contiguous or strided data. This benchmark is single threaded; however, we enabled MPICH asynchronous progress that triggers progress on communication in the background by forking an additional thread. Thus, when asynchronous progress is enabled, MPICH uses internally `MPI_THREAD_MULTIPLE` because two threads are running concurrently inside the runtime. We used three different communication modes:

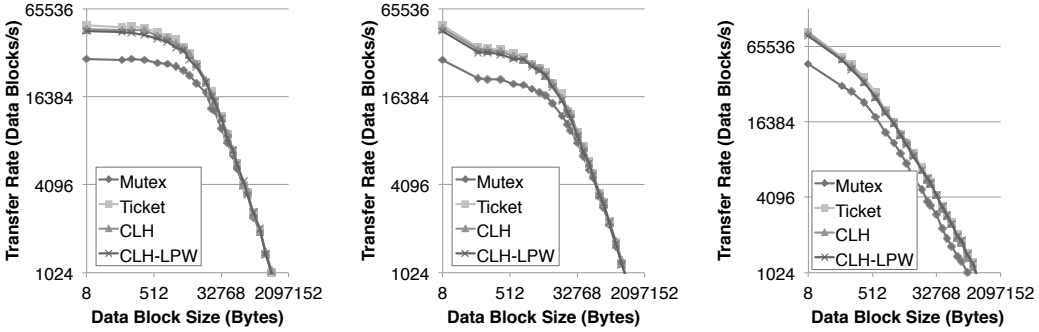
Contiguous. Data blocks are contiguous to each other.
Strided with direct mode. Data blocks are strided, and transfers are performed by generating datatypes and issuing a single operation.
Strided with vector mode. Data blocks are strided, and transfers are performed by issuing batches of operations.

The results of this experiment are shown in Figure 13 when running with 8 processes. We show results only with the `MPI_Accumulate` operation because `MPI_Get` and `MPI_Put` follow the same trend. Although only two threads are running concurrently, we notice a substantial performance difference between the Pthread mutex and the other locks. Specifically, the FIFO locks improve performance by up to 80% over the Pthread mutex. The reason is that the progress thread, which is most of the time in the progress loop, heavily monopolizes the lock because it does not do useful work most of the time. Thus, enforcing fairness produces a tremendous speedup. Similar to the two-sided communication results, the difference between the FIFO locks and CLH-LPW is not perceptible. Since only two threads are running concurrently, both the FIFO locks and CLH-LPW have the same arbitration property, while CLH-LPW adds negligible overhead because of its two-phase implementation. We observe that for strided data transfers in batch mode, improvements are visible even for large messages, because in this mode more operations are issued instead of packing them into one operation, which has the effect of entering the MPI runtime more frequently and exhibits more sensitivity to lock contention.

7.2. Kernels

In this section we evaluate some computational kernels often encountered in real applications.

7.2.1. Graph500 Benchmark BFS. The Graph500 benchmark is a communication-intensive code used for ranking large-scale systems in terms of graph-processing capabilities [Murphy et al. 2010]. It is composed of multiple kernels, but we consider only breadth-first search (BFS) in this work. More specifically, the baseline algorithm is an MPI-only level-synchronized BFS that relies on nonblocking point-to-point MPI communication for data exchanges. Our hybrid MPI+OpenMP implementation extends the MPI-only design by allowing multiple threads to cooperate for computation and independently communicate with remote processes. Moreover, both computation and communication are lock-free and atomic-free. Each thread maintains outgoing buffers corresponding to each remote process and one buffer for incoming messages. Threads repeatedly check for completed requests



(a) Contiguous (b) Strided with direct mode (c) Strided with batch mode

Fig. 13. Performance comparison of all methods when doing RMA contiguous data transfer using ARMCI-MPI with asynchronous progress.

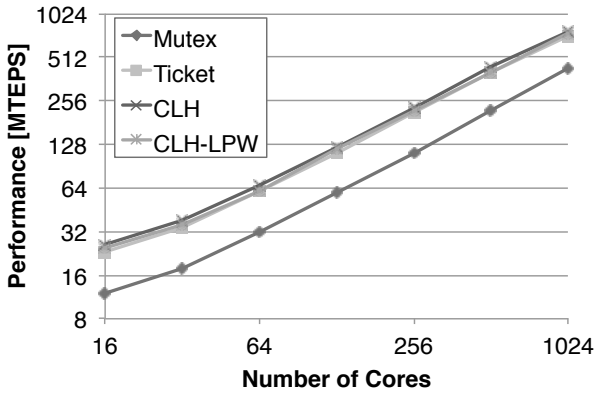
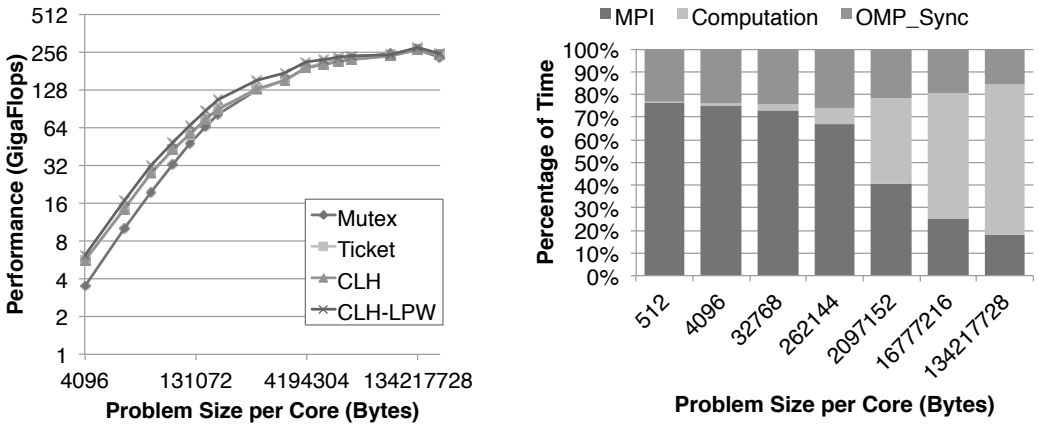


Fig. 14. Performance comparison of all methods with the Graph500 BFS kernel. Weak-scaling performance with one process per node, 8 threads per process, and problem sizes from 25 to 32.

using `MPI_Test` and eventually do computation and generate new outgoing requests. For a more detailed description, the reader can refer to [Amer et al. 2015b].

Figure 14 shows the results of a weak-scaling performance comparison of all the methods tested. We notice performance improvements of roughly 2x over the Pthread mutex. The best-performing lock is CLH with 78–117% improvement over the Pthread mutex. It also improves by 6–12% over the ticket lock. CLH-LPW does not show signs of superiority in this case. We explain this result by the fact that threads do not busy wait in the progress loop because they use only immediate `MPI_Test` calls to poll for progress. That is, all threads always have the same high priority inside the runtime, and CLH-LPW adds overhead of only around 2–10%.

7.2.2. 3D 7-Point Stencil Kernel. Stencil codes are a class of iterative methods found in many scientific and engineering applications. Here, the problem domain is iteratively updated by using the same computational pattern, called a *stencil*. We implemented a hybrid MPI+OpenMP 3D 7-point stencil code that simulates a heat equation. Our domain decomposition methodology tries to reduce the internode communication by dividing the domain along all dimensions to derive process subdomains. For thread blocks, we avoid splitting the



(a) 3D-stencil strong scaling

(b) 3D-stencil execution breakdown

Fig. 15. Strong-scaling comparison of all methods with the 3D 7-point stencil kernel. `OMP_Sync` refers to the time spent in the OpenMP synchronization barrier at the end of each iteration. It is also an indicator of load-unbalance.

process subdomain along the most strided dimensions for better cache performance. The basic communication method is to perform nonblocking send/receive operations at each iteration, followed by `MPI_Waitall` to wait for all the requests. Common hybrid stencil codes typically require `MPI_THREAD_FUNNELED` threading support, where the computation is done in parallel by a team of threads but only the main thread is driving the communication. In our implementation, all threads independently do computation and communication and synchronize only at the end of an iteration.

We conducted a strong-scaling experiment on 64 Nehalem nodes, using all 8 threads per node, while increasing the problem size. The results in Figures 15a and 15b show that our methods improve performance for relatively small problems (<= 4 MB per core). The reason is that, as demonstrated by our microbenchmarks, the runtime contention is more critical for small and medium-sized messages; in addition, the computation takes more time than does the communication, as shown in Figure 15b. Arguably, the benefit of our methods is less pertinent for stencil applications on this typical platform because bigger problems are often run in production. Nevertheless, given the increase in core counts and the trend of reduced memory per core, we expect that our runtime improvements will play a more important role when running stencil applications on platforms with less memory per core, such as current systems equipped with many-core accelerators and future systems. We observe that CLH-LPW performs the best, achieving up to 75% speedup over the Pthread mutex. We also note that CLH-LPW improves slightly over CLH (up to 17%). Here the benefit is limited because threads have few requests (8 receive and send request windows, where the maximum number of concurrent outstanding requests is 56) and the rate at which the critical section is entered from the main path is negligible compared with polling for communication in the progress loop. That is, most threads will fall back to low priority, which is equivalent to the FIFO locks. As a result, giving priority to the main path has limited benefits.

7.3. SNAP: A Particle Transport Proxy Application

SNAP [10] is a proxy application, or mini-app, that models the computational characteristics of the PARTISN transport code. PARTISN is an MPI parallel application that solves the linear Boltzmann transport equation on multi-dimensional grids. SNAP does not solve actual physical problems but rather approximates the computational intensity, memory footprint, and communication patterns of PARTISN. For a time-dependent transport equation, SNAP exploits parallelism at the level of the spatial, the angular, and the energy dimensions. Computation along the energy domain is carried out typically through several energy groups, which are suitable for thread-level parallelism of modern cluster nodes. Because of its prohibitive size, the spatial domain is partitioned across MPI ranks and traversed through sweeps along the discrete direction of the angular domain with the necessary data exchanges between ranks. The traversal follows the parallel KBA wavefront method. Most of the communication is performed using point-to-point communication. Furthermore, threads are configured to perform both computation and MPI communication concurrently.

For our evaluation, we used one of the regression tests that uses the method of manufactured solutions setting to generate a source for the SNAP computations in a three dimensional spatial domain. This test was used as a base to generate scaling experiments. The full list of input parameters used in this evaluation are shown in Table II. The parameters we vary in our scaling experiments are the spatial dimensions along the y and z dimensions (`ny` and `nz`) and the corresponding number of MPI ranks along each dimension (`npey` and `npez`). For the strong scaling experiments, we fix the problem size to (`nx`, `ny`, `nz`) = (128,16,16), to fit in the memory of a single node, and vary the number of MPI ranks by doubling the number of ranks per one of the y or z dimensions while alternating. For the weak-scaling experiment, we start with a problem (128,4,4), and increase the number of MPI ranks and the problem size only on the y dimension, which showed the most insightful results [11]. In both experiments, we spawn one MPI rank per NUMA node and four threads per MPI rank.

The scaling results on the Nehalem cluster are shown in Figure 16. In the strong scaling case, we observe linear scaling for small core counts independently of the locking implementation. From 128 cores onwards, however, we notice losses in scalability with the Pthread mutex based implementation being the most suffering and being more than 350% slower than the CLH-based implementation. These results imply that SNAP shifts to a communication intensive regime that requires a scalable multithreaded MPI implementation to sustain acceptable performance at large scale. In the weak-scaling experiment, the problem size per MPI rank on the y and z dimensions was kept low and constant to stress MPI communication. In Figure 16b, we observe that the lock implementation plays an important role in mitigating communication overheads. Going from two to 128 nodes, the Pthread mutex implementation suffers almost a 10-fold performance drop while the other implementations suffer drops below 3-fold. Furthermore, we observe up to 10-fold performance gap between the Pthread mutex and the lock implementations on 64 nodes.

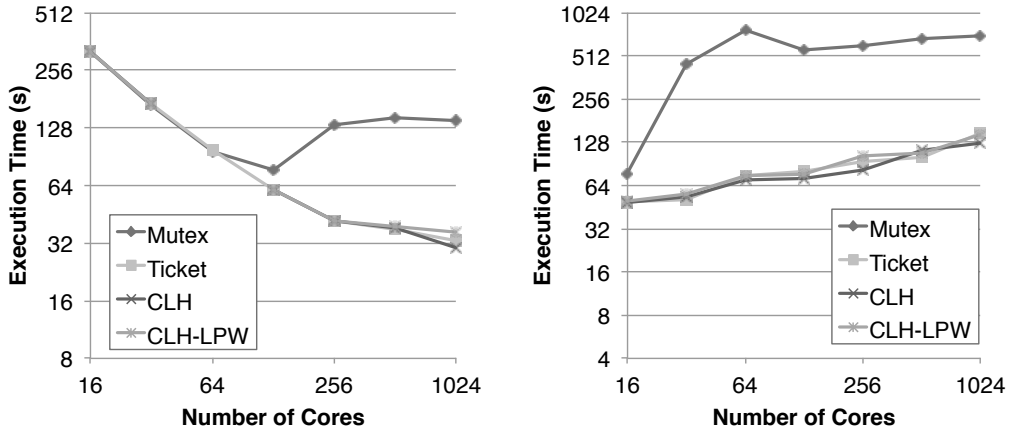
The large performance gap between using Pthread mutex and the other locks is due to a combination of fine-grained communication, strong communication dependencies that are tied to the wavefront communication pattern, and using message tags in the point-to-point communication. First, fine-grained communication originates from our choice of reducing the problem size on the y and z dimensions to stress interprocess communication, since the domain decomposition follows these same dimensions. These input parameters not only stress communication and thread synchronization within the MPI runtime, but also emulate workloads on future systems that will likely feature more limited memory capacity

[10] https://github.com/losalamos/snap

[11] Other variations in the input parameters and scaling methods showed whether no contention for MPI communication, or scalability bottlenecks that are not strongly related to locking issues.

Table II. SNAP input parameters

nthreads	4	lx	0.08	src_opt	3	iitm	5
npey	2	lz	0.08	timedep	1	epsi	0.0001
npez	2	ly	0.08	it_det	0	scatp	0
ndimen	3	nmom	4	tf	1.0	fixup	0
nx	128	nang	32	nsteps	10	angcpy	1
ny	16	ng	64	oitm	40	ichunk	4
nz	16	mat_opt	1	fluxp	0		



(a) Strong scaling (b) Weak scaling

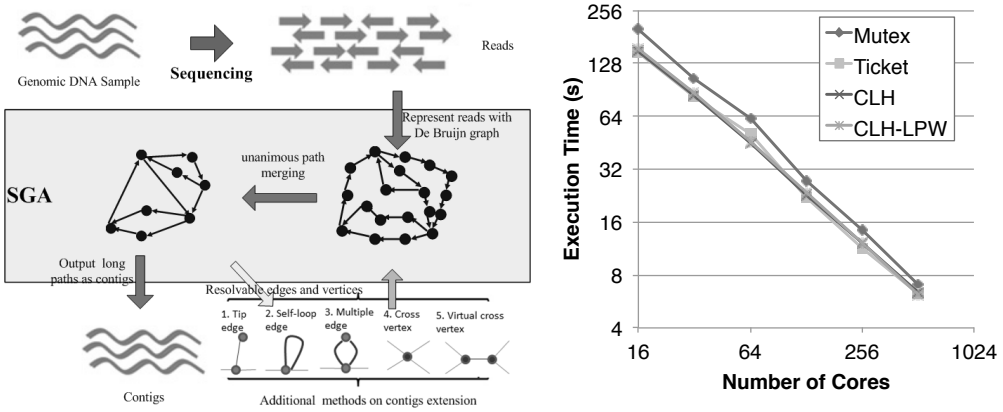
Fig. 16. SNAP scaling on the Nehalem cluster. We only show the time for the *Transport Sweep* step, which is the most time consuming.

per core and massive core counts. Second, the strong data dependencies also exacerbated the overhead of lock monopolization. If we model the communication pattern as a directed graph, where edges represent interprocess data dependencies, slowing an edge of the graph (e.g. waiting threads delay threads issuing send operations) will propagate though the graph. Consequently, the overhead of the lock monopolization grows with the complexity of the dependency graph. Finally, the message tags are used to correctly match the buffer offset within the boundary data and avoid mismatches. This causes strict matching patterns that suffer from lock monopolization as demonstrated in the previous sections. All these issues were reproduced in our microbenmarks and confirm that they can truly occur in real applications.

7.4. Genome Assembly Application

Genome assembly is an important process for many fields, such as biological research and virology. It refers to the process of reconstructing a long DNA sequence, such as the chromosome of an organism, from a set of *reads* (short DNA sequences) by aligning and merging them. Automated sequencing machines can generate billions of reads to be processed by assembly applications. These applications exploit high-performance computing systems and explore efficient parallel solutions in order to cope with the ever-increasing sequencing data being generated.

We evaluated our methods using the SWAP-Assembler [Meng et al. 2014], an application that targets processing massive sequencing data on large-scale parallel architectures. It abstracts the genome assembly problem with a multistep bidirected graph and relies on a scalable framework, SWAP (Small World Asynchronous Parallel) [Meng et al. 2012], to



(a) Genome assembly process in SWAP-Assembler (b) SWAP-Assembler strong scaling

Fig. 17. Genome assembly process and performance of the SWAP-Assembler using all methods.

perform computational work in parallel (Figure 17a). The SWAP-Assembler takes the k-mer length as a parameter during the De Bruijn graph construction. The SWAP framework is implemented on top of MPI to ensure interprocess communication. Each process spawns two threads, one for sending and another for receiving data from other processes using blocking `MPI_Send`/`MPI_Recv` operations. We performed a strong-scaling experiment with a synthetic sequence of 1 million reads, where each read contains 36 nucleotides while using k-mer lengths of 21. The results are shown in Figure 17b. For each data point we used four processes per node and two threads per process to utilize all cores. We observe that the user-space locks perform similarly and outperform the Pthread mutex by 10% to 38%. We note that this improvement in processing time does not incur any modification in the application or the underlying hardware. This fact is important for applications in production environments because no additional investment is required to speed the time to solution.

8. DISCUSSION

In our introduction to thread safety in Section 3, we considered a critical-section granularity as orthogonal to how it is arbitrated. That is, regardless of the granularity, resource acquisition–related issues such as starvation may occur, and appropriate arbitration methods will be required. Thus, we believe that combining those approaches will have a synergistic effect on reducing the runtime contention. However, a cost-effectiveness study of both methods on the same testbed is still warranted. Such a study can guide the development process of a thread-safe library. A possible model would be to start with a global critical section, explore effective arbitration methods, reduce granularity if high contention persists, and repeat the process.

In addition to the lower overhead of the CLH lock compared with the the CLH-LPW lock (fewer atomic operations), cases may arise where the FIFO locks will perform better because of the FIFO arbitration. MPI runtimes are sensitive to the number of queued requests because the associated internal data structures and algorithm complexity are proportional. Since the CLH-LPW prioritizes feeding the runtime with requests, the FIFO locks may reduce the rate of issuing requests and thus their associated overhead. A related issue exists, however, that affects the cache performance of the MPI runtime and applications. Specifically, the order in which threads acquire resources can affect the data locality of MPI internal structures, such as shared queues, and thus affect the runtime performance. Similarly, the order can affect the computation part of an application. Assuming that the

aggregated threads working sets cannot all fit in the last level of cache, critical section arbitration might affect the amount of data reuse and cache line evictions. These intricacies require further analysis and experimentation.

The best-performing implementations in this study rely on FIFO locks, while the CLH queue lock outperforms the ticket lock because of its relatively lower hand-off latency. Intuitively, one would expect that the best heuristic for improving communication progress would involve circulating the lock looking for a thread that would make progress. Decreasing the cost of the hand-off further while conserving the same arbitration is difficult, however. If more effort is invested in decreasing the hand-off latency to the detriment of the arbitration, it might not lead to improved communication progress. For instance, a cohort or hierarchical lock will reduce the hand-off cost by circulating the lock in the local memory domain. This active set of threads is not guaranteed to do any work and might hurt the overall progress. Leveraging high-throughput locks and a suitable arbitration is a topic that we are currently exploring. Our early experiments with a two-level cohort CLH lock showed message throughputs similar to that of our priority CLH-ticket lock.

9. CONCLUSION AND FUTURE WORK

In this work we considered locking aspects in multithreaded MPI implementations. We first investigated the most widely used lock, the Pthread mutex. Our analysis showed that contention and unbounded lock monopolization take place within the MPI runtime. We demonstrated how these affect communication progress by causing inadequate arbitration within the MPI runtime. By introducing fairness and by biasing the arbitration toward the benefit of communication progress, we obtained substantial improvement with several benchmarks and applications.

Despite those benefits, the throughput difference between our improved MPI runtime and a lower messaging layer suggests that room for improvement remains. We are currently investigating even more flexible methods that target reducing wasted time by the threads inside the MPI runtime. One example is selective thread wakeup triggered by events such as message arrival. We are also considering combining fine-grained critical sections with custom lock arbitrations.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 23220003 and by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

REFERENCES

Abdelhalim Amer, Pavan Balaji, Wesley Bland, William Gropp, Rob Latham, Huiwei Lu, Lena Oden, Antonio Pena, Ken Raffenetti, Sangmin Seo, and others. 2015a. MPICH User's Guide. (2015).

Abdelhalim Amer, Huiwei Lu, Pavan Balaji, and Satoshi Matsuoka. 2015b. Characterizing MPI and Hybrid MPI+Threads Applications at Scale: Case Study with BFS. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 1075–1083.

Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Pavan Balaji, and Satoshi Matsuoka. 2015c. MPI+Threads: runtime contention and remedies. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 239–248.

Pavan Balaji, Darius Buntinas, D. Goodell, W. D. Gropp, and Rajeev Thakur. 2010. Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming. *Int. J. High Perform. Comput.Appl.* 24 (Feb. 2010), 49–57.

Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High Performance Locks for Multi-Level NUMA Systems. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 215–226.

Travis Craig. 1994. Building FIFO and Priority Queuing Spin Locks from Atomic Swap. (1994).

James Dinan, Pavan Balaji, Dave Goodell, Doug Miller, Marc Snir, and Rajeev Thakur. 2013. Enabling MPI Interoperability through Flexible Communication Endpoints. In *EuroMPI 2013*. Madrid, Spain, 13–18.

James Dinan, Pavan Balaji, Jeff R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. 2012. Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication. In *2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*. 739–750.

Gábor Dózsa, Sameer Kumar, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Joe Ratterman, and Rajeev Thakur. 2010. Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface (EuroMPI'10)*. Springer-Verlag, Berlin, Heidelberg, 11–20.

Ulrich Drepper. 2005. Futexes Are Tricky. (Dec. 2005).

Hubertus Franke, Rusty Russell, and Matthew Kirkwood. 2002. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *AUUG Conference Proceedings*. 85.

David Goodell, Pavan Balaji, Darius Buntinas, Gabor Dozsa, William Gropp, Sameer Kumar, Bronis R. de Supinski, and Rajeev Thakur. 2010. Minimizing MPI Resource Contention in Multithreaded Multicore Environments. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing (CLUSTER '10)*. IEEE Computer Society, Washington, DC, USA, 1–8.

William Gropp and Rajeev Thakur. 2007. Thread-Safety in an MPI implementation: Requirements and Analysis. *Parallel Comput.* 33 (Sept. 2007), 595–604.

Torsten Hoefler, Greg Bronevetsky, Brian Barrett, Bronis R. de Supinski, and Andrew Lumsdaine. 2010. Efficient MPI Support for Advanced Hybrid Programming Models. In *Recent Advances in the Message Passing Interface (EuroMPI'10), Lecture Notes in Computer Science*, Vol. 6305. Springer, 50–61.

John M Mellor-Crummey and Michael L Scott. 1991a. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 9, 1 (1991), 21–65.

John M. Mellor-Crummey and Michael L. Scott. 1991b. Synchronization without Contention. In *ACM SIGARCH Computer Architecture News*, Vol. 19. ACM, 269–278.

Jintao Meng, Bingqiang Wang, Yanjie Wei, Shengzhong Feng, and Pavan Balaji. 2014. SWAP-Assembler: Scalable and Efficient Genome Assembly towards Thousands of Cores. *BMC Bioinformatics* 15, Suppl 9 (2014), –2.

Jintao Meng, Jianrui Yuan, Jiefeng Cheng, Yanjie Wei, and Shengzhong Feng. 2012. Small World Asynchronous Parallel Model for Genome Assembly. In *Network and Parallel Computing, Lecture Notes in Computer Science*, James J. Park, Albert Zomaya, Sang-Soo Yeo, and Sartaj Sahni (Eds.). Vol. 7513. Springer Berlin Heidelberg, 145–155.

Ingo Molnar. 2003. *The Native POSIX Thread Library for Linux*. Technical Report.

Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. 2010. Introducing the Graph 500. *Cray Users Group (CUG)* (2010).

Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. 2006a. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Int. J. High Perform. Comput. Appl.* 20, 2 (May 2006), 203–231.

Jarek Nieplocha, Vinod Tipparaju, Manojkumar Krishnan, and D. K. Panda. 2006b. High Performance Remote Memory Access Communication: The ARMCI Approach. *Int. J. High Perform. Comput. Appl.* 20, 2 (May 2006), 233–253.

Marat Valiev, Eric J. Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P. Straatsma, Hubertus J. J. Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L. Windus, and Wibe A. de Jong. 2010. NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations. *Computer Physics Communications* 181, 9 (2010), 1477–1489.